SOFTWARE VERIFICATION AND VALIDATION
Evaluation of Fault Detection Effectiveness for Combinatorial and Exhaustive Selection of Discretized Test Inputs

Carmelo Montanez, D. Richard Kuhn, Mary Brady, Richard M. Rivello, Jenise Reyes, and Michael K. Powers, National Institute of Standards and Technology

Testing components of Web browsers and other graphical interface software can be extremely expensive because of the need for human review of screen appearance and interactive behavior. Combinatorial testing has been advocated as a method that provides strong fault detection with a small number of tests, although some authors have disputed its effectiveness. This article compares the effectiveness of combinatorial test methods with exhaustive testing of discretized inputs for the document object model events standard. More than 36,000 tests – all possible combinations of equivalence class values – were reduced by more than a factor of 20 with an equivalent level of fault detection, suggesting that combinatorial testing is a cost-effective method of assurance for Web-based interactive software.

Key words: combinatorial testing, conformance testing, document object model, interoperability testing, World Wide Web standards

INTRODUCTION

Test input selection is a critical task in software testing, because it is generally impossible to test all possible combinations of inputs, particularly for continuous-valued variables. Representative discrete values for input variables must be chosen using some form of category partitioning (Ammann and Offutt 2008). After inputs are discretized, there still remains the task of selecting inputs for tests to be applied to the system. Possibilities for input selection include ad hoc, random, each-choice, pairwise, and generalized *t*-way testing. For ad hoc testing, judgment may be used to determine test inputs that the tester believes are most critical or likely to detect errors. Random testing requires sampling input values according to some distribution and sampling level. The other strategies listed previously can be defined for a set of N input variables, with v_1, v_2, …, v_N values per variable, by specifying coverage of t-way combinations in the full test set as follows: each-choice means that t=1 and every variable value is included in some test; for pairwise, every two-way combination is covered, and t-way testing for $t > 2$ means that every t-way combination is covered at least once.

Combinatorial or t-way testing is among the test approaches that appear to offer good fault detection with a small test set (Grindal, Offutt, and Andler 2005), including its simplest form, pairwise (t=2) testing (Lei and Tai 1998; Tai and Lei 2002). Because system failures often result from the interaction of conditions that might be innocuous individually, this method can be effective for domains with many interacting parameters, such as interoperability testing. Consider a large example: a manufacturing automation system that has 20 controls, each with 10 possible settings, has a total of 10^{20} combinations. Surprisingly, one can check all pairs of these values with less than 200 tests, if the tests are carefully constructed. Pairwise testing has become popular because it can check for problem-causing interactions with relatively few tests. Several investigations suggest individual values or a pair of parameters are responsible for roughly 70 percent to more than 97 percent of faults (Kuhn and Reilly 2002; Kuhn, Wallace, and Gallo 2004).

Empirical results suggest that extended forms of combinatorial testing, covering combinations beyond simple pairwise, can be as effective as testing all possible combinations (Kuhn, Wallace, and Gallo 2004; Bell 2006), because if all faults are triggered by interactions of one to six variables, then testing all six-way combinations can provide a high degree of confidence. Some authors, however, argue that combinatorial or t-way testing may be no more effective than other approaches (Bach and Schroeder 2004; Jorgensen 2008). In this article, the authors compare the fault detection effectiveness of *t*-way testing with full exhaustive testing of discretized inputs for implementations of the document object model (DOM) events standard. Because testing DOM events requires substantial human involvement, testing can be extremely time consuming and expensive. Thus, there is a need for methods to reduce the number of test inputs while retaining a high level of fault detection. As described in the next section, the DOM test suite had already been applied with exhaustive (with respect to discretized values) tests against a variety of commercial DOM implementations, so it provided a valuable opportunity to evaluate the hypothesis that combinatorial testing could provide an equal or better level of fault detection as exhaustive testing, using fewer tests. If results showed that a much smaller test suite could achieve the same level of fault detection as exhaustive tests, then conformance testing could be done at a much lower cost in staff time and resources.

THE DOCUMENT OBJECT MODEL

The DOM (W3C 2011) is a standardized method for representing and interacting with components of XML, HTML, and XHTML documents. DOM lets programs and scripts access and update the content, structure, and style of documents dynamically, making it easier to produce Web applications in which pages are accessed nonsequentially. DOM is standardized by the World Wide Web Consortium (W3C).

Since its origination in 1996 as a convention for accessing and modifying parts of Javascript Web pages (known now as DOM Level 0), DOM has evolved as a series of standards offering progressively greater capabilities. Level 1 introduced a model that allowed changing any part of the HTML document, and Level 2 added support for XML namespaces, load and save, cascading style sheets (CSS), traversing the document, and working with ranges of content. Level 3 brings additional features, including keyboard event handling.

DOM Level 3 Events (W3C 2009) is a W3C standard developed by the Web Applications Working group. Implemented in browsers, it is a generic platform and language neutral event system that allows registration of event handlers, describes event flow through a tree structure, and provides basic contextual information for each event. This work builds on the previous DOM Level 2 events specifications. There are two basic goals in the design of DOM Level 3 events. The first goal is to design an event system that allows registration of event listeners and describes an event flow through a tree structure. The second goal is to provide a common subset of the current event system used on DOM Level 3 events browsers.

DOM browser implementations typically contain tens of thousands of source lines of code. To help ensure successful implementations of this complex standard, NIST developed the DOM conformance test suites, which include tests for many DOM components. Early DOM tests were hand-coded in a test language, then processed to produce ECMAScript and Java. In the current version of the test suites, tests are specified in an XML grammar, allowing easy mapping from specification to a variety of language bindings. Because the grammar is generated automatically from the DOM specs, tests can be constructed quickly and correctly. Output of the test generation process includes the following components, which implementers can use in testing their product for DOM interoperability:

- Tests in the XML representation language
- XSLT stylesheets necessary to generate the Java and ECMAScript bindings

- Generated executable code

To reduce the time needed to generate the large number of tests required for checking standards conformance, NIST developed a test accelerator tool (NIST 2011) that was used to generate tests for 35 (out of 36) DOM events. The specification defines each event as an interface definition language (IDL), which in turn defines a number of functions for each event. A typical function can have anywhere from one to 15 parameters. Since the IDL definition could be accessed directly from the specs website, the Web address was given as input to the Java application. This way the application could read and traverse them extracting just the information of interest. In this case, the function names and their respective parameters, argument names, and so on, became part of the XML file that was used to feed the test accelerator to automatically create the DOM Level 3 tests.

Category partitioning was used to select representative values for non-Boolean parameters. The initial test set was exhaustive across the equivalence classes, producing 36,626 tests that exercised all possible combinations of representative parameter values. Three different implementations were tested. The implementations successfully executed about 48.49 percent of the test cases and generated a total of 10 distinct messages that indicated a test could not be run because of a problem such as a nonsupported feature. The DOM events and number of tests for each are shown in Table 1. This set of exhaustive tests detected a total of 72 failures. Test suites of this size are not uncommon for significant real-world software. For example, the W3C test suites for XML include 40,000 for XML Schema, 17,487 for XML Query, and 3,366 for Core, a total of more than 60,000 tests for the XML based standards alone. A survey by the publisher O’Reilly found that 11 percent of test practitioners were using test suites exceeding 10,000 tests

(O’Reilly 2008). Thus, the results reported in this article may have broad application in practical software and systems testing.

COMBINATORIAL TESTING OF DOM EVENTS

To investigate the effectiveness of combinatorial testing, *covering arrays* of two-way through six-way tests were produced. A covering array defines a set of tests that cover all *t*-way combinations in a highly compact form. A variety of high-quality free tools are available for producing covering arrays, including Microsoft PICT and ACTS, developed by the National Institute of Standards and Technology and the University of Texas Arlington. Using *t*-way combinations can significantly reduce the number of tests as compared with exhaustive. For example, the *mousedown* event (see Figure 1) requires 4352 tests if all combinations are to be realized. Combinatorial testing reduces the set to 86 tests for four-way coverage. An excerpt of these tests is shown in Figure 1 (function arguments are: `'type', bubbles, cancelable, windowObject, detail, screenX, screenY, clientX, clientY, ctrlKey, altKey, shiftKey, metaKey, button, relatedTarget).` Note that the covering array tool is independent of the test accelerator tool described in the previous section, and can be used without the test accelerator. A variety of studies have investigated the effectiveness of combinatorial testing (Cohen, Snyder, Rothermel 2004; duBousquet et al. 2004; Kuhn et al. 2009), although none prior to this have compared this method with exhaustive testing.

Table 2 details the number of parameters and number of tests produced for each of the 35 DOM events, for *t* = 2 through 6. That is, the tests covered all two-way through six-way combinations of values. Note that for events with few parameters, the number of tests is the same for the original test suite (see Table 1) and combinatorial for various levels of t. For example, 12 tests were produced for Abort in the original and also for combinatorial testing at *t* = 3 to 6. This is because producing all *n*-way combinations for *n* variables is simply all possible combinations of these *n* variables, and Abort has three variables. This situation is not unusual when testing configurations with a limited number of values for each parameter. For nine of the 35 events (two click events, six mouse events, and wheel), all combinations are not covered even with six-way tests. For these events, combinatorial testing provides a significant gain in efficiency (see Table 2).

```
1  "mousedown" true true window 5 5 5 5 5 true true true true 5 null
2  "mousedown" true true window 5 5 5 5 5 true true true true 10 null
3  "mousedown" true true window 5 5 5 5 5 true true true false 5 null
4  "mousedown" true true window 5 5 5 5 5 true true false true 5 null
5  "mousedown" true true window 5 5 5 5 5 true true false true 5 null
6  "mousedown" true true window 5 5 5 5 5 true true false true 10 null
      ...
83 "mousedown" true true window 5 5 5 -5 5 false true true false 5 null
84 "mousedown" true true window 5 5 5 -5 5 false true true false 10 null
86 "mousedown" true true window 5 5 5 -5 5 false true false true 5 null
86 "mousedown" true true window 5 5 5 -5 5 false true false true 10 null
```

Figure 1 Excerpt of 86 combinatorial tests produced for “mousedown” event.

TEST RESULTS

Table 3 shows the faults detected for each event. All conditions flagged by the exhaustive test suite were also detected by three of the combinatorial testing scenarios (four-, five-, and six-way testing), which means that the implementation faults were triggered by four-way interactions or less. Pairwise testing would have been inadequate for the DOM implementations, because two-way and three-way tests detected only 37.5 percent of the faults. As can be seen in Table 3, the exhaustive (all possible combinations) and the four-way to six-way combinatorial tests were equally successful in fault detection, indicating that exhaustive testing added no benefit beyond four-way tests. These findings are consistent with the studies described earlier in this article, which showed that software faults tend to be triggered by interactions of no more than six variables, for the applications studied so far. Using combinatorial methods, one is able to take advantage of this finding and limit the size of conformance test suites, greatly reducing costs. DOM testing was somewhat unusual in that exhaustive testing was possible at all. For most software, too many possible input combinations exist to cover even a tiny fraction of the exhaustive set, so combinatorial methods may be of greater benefit for these.

The exhaustive approach used a total of 36,626 tests (see Table 1) for all combinations of events, but after applying combinatorial testing, the set of tests is dramatically reduced, as shown in Table 3. The number of tests generated in combinatorial covering arrays is proportional to v^t log n, for t-way interactions where each of n parameters has v values. In cases where most parameters have a small number of discrete values, such as DOM events, this is less of a limitation, but it was required for parameters such as screen X and Y values, and must be considered for most software testing.

Table 3 shows results for two-way through six-way testing. An interesting observation that can be gathered by examining the data is that although the number of tests that successfully execute varies from t-way combination to t-way combination, the number of failures remains a constant at t = 2 and 3, and at t = 4 to 6. The last column shows the tests that did not execute to completion, in almost all cases due to nonsupport of the feature under test.

DOM results were consistent with previous findings that testing a small number of interactions (in this case four-way) was sufficient to detect all errors. Comparing results of the DOM testing with previously reported data on t-way interaction failures, one can see that some

DOM failures were more difficult to detect, in the sense that a smaller percentage of the total were found by three-way tests than for the other application domains, where testing through three-way combinations typically detected more than 80 percent of faults (Kuhn, Wallace, and Gallo 2004). The unusual distribution of fault detection for DOM tests may result from the large number of parameters for which exhaustive coverage was reached (so the number of tests remained constant after a certain point). There are thus two sets of events: a large set with few possible values that could be covered exhaustively with two-way or three-way tests, and a smaller set with a larger input space (from 1024 to 4352). In particular, nine events (click, dblClick, mouse events, and wheel) all have the same input space size, with number of tests increasing at the same rate for each, while for the rest, exhaustive coverage is reached at either t=2 or t=3. The ability to compare results of previously conducted exhaustive testing with combinatorial testing provides an added measure of confidence in the applicability of these methods to this type of interoperability testing.

CONCLUSIONS

The DOM events testing suggests that combinatorial testing can significantly reduce the cost and time required for conformance testing for Web standards with characteristics similar to DOM. What is the appropriate interaction strength to use in this type of testing? Intuitively, it seems that if no additional faults are detected by t-way tests, then it may be reasonable to conduct additional testing only for t+1 interactions, but no greater if no additional faults are found at t+1. In empirical studies of software failures, the number of faults detected at $t > 2$ decreased monotonically with t, and the DOM testing results are consistent with this earlier finding. Following this strategy for the DOM testing would result in running two-way tests through five-way, then stopping because no additional faults were detected beyond the four-way testing. Alternatively, given the apparent insufficient fault detection of pairwise testing, testers may prefer to standardize on a higher level of interaction coverage, say three-way or four-way. This option may be particularly attractive for an organization that produces a series of similar products and has enough experience to identify the most cost-effective level of testing. Even the relatively strong four-way testing in this example was only 5 percent of the original test set size.

What is the best strategy for applying combinatorial methods to interoperability testing? This question can be investigated in future applications of combinatorial methods. Results in this study have been sufficiently promising for combinatorial methods to be applied in testing other interoperability standards.

REFERENCES

Ammann, P., and J. Offutt. 2008. *Introduction to software testing*. New York: Cambridge University Press.

Bach, J., and P. Shroeder. 2004. Pairwise testing - A best practice that isn't. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*, 180-196

Bell, K. Z. 2006. Optimizing effectiveness and efficiency of software testing: A hybrid approach. PhD diss., North Carolina State University.

Cohen, D. M., S. R. Dalal, J. Parelius, and G. C. Patton. 1996. The combinatorial approach to automatic test generation. *IEEE Software* 13, no. 5 (September):83-88.

Cohen, M. B., J. Snyder, and G. Rothermel. 2004. Testing across configurations: Implications for combinatorial testing. In *Proceedings of the Workshop on Advances in Model-Based Software Testing*, IEEE Press, 1–9.

du Bousquet, L., Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. 2004. A case study in JML-based software validation. In *Proceedings of 19th International IEEE Conference on Automated Software Engineering*, 294-297, Linz.

Grindal, M., J. Offutt, and S. F. Andler. 2005. Combination testing strategies: A survey. *Journal of Software Testing, Verification and Reliability* 15, no. 3:167-199.

Jorgensen, P. C. 2008. *Software testing: A craftsman's approach, third edition*, Auerbach Publications.

Kuhn, D. R., and M. J. Reilly. 2002. An investigation of the applicability of design of experiments to software testing. *27th NASA/IEEE Software Engineering Workshop*, IEEE Computer Society, 91-95, 4-6 December.

Kuhn, D. R., D. Wallace, and A. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, no. 6:418-421.

Kuhn, R., R. Kacker, Y. Lei, and J. Hunter. 2009. Combinatorial software testing. *IEEE Computer* 42, no. 8 (August).

Lei, Y., and K. C. Tai. 1998. In-parameter order: A test generation strategy for pairwise testing. In *Proceedings of the Third IEEE High Assurance Systems Engineering Symposium*, 254-261, IEEE, November.

National Institute of Standards and Technology (NIST). 2011. Test accelerator. http://www.itl.nist.gov/div897/docs/testacc.html

O'Reilly. 2008. Survey: About your test suites. Available at:

http://www.perlmonks.org/?displaytype=print;node_id=701817 .

Tai, K. C., and Y. Lei. 2002. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 28, no. 1 (January):109-111.

W3C. 2009. World Wide Web Consortium. 2009. DOM Level 3 Events Specification. Available at: http://www.w3.org/TR/DOM-Level-3-Events/ .

W3C. 2011. World Wide Web Consortium. Document object model. Available at: http://www.w3.org/DOM/ .

BIOGRAPHIES

Carmelo Montanez is a computer scientist at the National Institute of Standards and Technology (NIST) in Gaithersburg, MD. His main interest is the Conformance Testing area, specifically generating tests automatically. Carmelo has been involved with many XML technologies including DOM, XSL Formatting Objects, XSLT, and XML Query. Carmelo is currently working on developing a schema for Computer Forensics. He received a BS in Mathematics and Computer Sciences and an AA in Chemistry from the University of Puerto Rico.

Rick Kuhn is a computer scientist in the Computer Security Division of the National Institute of Standards and Technology. He has authored more than 100 publications on information security, empirical studies of software failure, and software assurance, currently focusing on combinatorial testing. He co-developed the role based access control model (RBAC) used throughout industry and led the effort that established RBAC as an ANSI standard. He received an MS in computer science from the University of Maryland College Park, and a BA and MBA from the College of William & Mary.

Mary Brady is the Manager of the Information Systems Group of the National Institute of Standards and Technology. Over the last decade, she has led multiple XML-based testing efforts, leading to tens of thousands of conformance tests that resulted from increasing levels of automatic test generation. She earned a MS in Computer Science from George Washington University, and a BS in Computer Science and Mathematics from Mary Washington College.

Richard Rivello is a computer scientist in the Software and Systems Division of the National Institute of Standards and Technology (NIST). One of the Software and Systems Division’s missions is to advance the state of the art of software testing by developing scientifically rigorous, breakthrough techniques to automatically generate tests that are cheaper to develop and more comprehensive. Mr. Rivello has a wide range of experience in testing various W3 standards such as XML, Document Object Model (DOM) Levels 1 and 2 Events. Mr. Rivello has a Bachelor of Science degree in Computer Science from Youngstown State University (1984).

Jenise Reyes-Rodriguez is a computer scientist at the National Institute of Standard and Technology in Gaithersburg, MD. Her previous work was in the area of Conformance Testing, which included the DOM Level 3, XQuery and Mobile Web standards from the W3C. She is currently working in Computer Forensics. She has a BS in Mathematics and Computer Sciences from the University of Puerto Rico.

Michael Kishi Powers is a junior at the University of Maryland, College Park. He is majoring in Electrical Engineering and is minoring in Astronomy. He has been an intern at the National Institute of Standards and Technology since 2008.

Event name	Number of parameters	Number of tests
Abort	3	12
Blur	5	24
Click	15	4352
Change	3	12
dblClick	15	4352
DOMActivate	5	24
DOMAttrModified	8	16
DOMCharacterDataModified	8	64
DOMElementNameChanged	6	8
DOMFocusIn	5	24
DOMFocusOut	5	24
DOMNodeInserted	8	128
DOMNodeInsertedIntoDocument	8	128
DOMNodeRemoved	8	128
DOMNodeRemovedFromDocument	8	128
DOMSubTreeModified	8	64
Error	3	12
Focus	5	24
KeyDown	1	17
KeyUp	1	17
Load	3	24
MouseDown	15	4352
MouseMove	15	4352
MouseOut	15	4352
MouseOver	15	4352
MouseUp	15	4352
MouseWheel	14	1024
Reset	3	12
Resize	5	48
Scroll	5	48
Select	3	12
Submit	3	12
TextInput	5	8
Unload	3	24
Wheel	15	4096
Total Tests		36626

Table 1 DOM Level 3 events tests – exhaustive

Event name	Num param	2-way tests	3-way tests	4-way tests	5-way tests	6-way tests
Abort	3	8	12	12	12	12
Blur	5	10	16	24	24	24
Click	15	18	40	86	188	353
Change	3	8	12	12	12	12
dblClick	15	18	40	86	188	353
DOMActivate	5	10	16	24	24	24
DOMAttrModified	8	8	16	16	16	16
DOMCharacterDataModified	8	32	62	64	64	64
DOMElementNameChanged	6	8	8	8	8	8
DOMFocusIn	5	10	16	24	24	24
DOMFocusOut	5	10	16	24	24	24
DOMNodeInserted	8	64	128	128	128	128
DOMNodeInsertedIntoDocument	8	64	128	128	128	128
DOMNodeRemoved	8	64	128	128	128	128
DOMNodeRemovedFromDocume	8	64	128	128	128	128
DOMSubTreeModified	8	32	64	64	64	64
Error	3	8	12	12	12	12
Focus	5	10	16	24	24	24
KeyDown	1	9	17	17	17	17
KeyUp	1	9	17	17	17	17
Load	3	16	24	24	24	24
MouseDown	15	18	40	86	188	353
MouseMove	15	18	40	86	188	353
MouseOut	15	18	40	86	188	353
MouseOver	15	18	40	86	188	353
MouseUp	15	18	40	86	188	353
MouseWheel	14	16	40	82	170	308
Reset	3	8	12	12	12	12
Resize	5	20	32	48	48	48
Scroll	5	20	32	48	48	48
Select	3	8	12	12	12	12
Submit	3	8	12	12	12	12
TextInput	5	8	8	8	8	8
Unload	3	16	12	24	24	24
Wheel	15	20	44	92	214	406
Total Tests		702	1342	1818	2742	4227

Table 2 DOM 3 level tests - combinatorial

t-way combinations	Number of tests	Pct of exhaustive	Passed	Failed	Not executed
2 Way	702	1.92%	202	27	473
3 Way	1342	3.66%	786	27	529
4 Way	1818	4.96%	437	72	1309
5 Way	2742	7.49%	908	72	1762
6 Way	4227	11.54%	1803	72	2352
Exhaustive	36,626		29,218	72	7336

Table 3 Results for all t-way combinations